5-30-2018

Poster: Towards safe refactoring for intelligent parallelization of Java 8 streams

Yiming Tang
CUNY Graduate Center

Raffi T. Khatchadourian
CUNY Hunter College

Mehdi Bagherzadeh
Oakland University

Syed Ahmed
Oakland University

Towards Safe Refactoring for Intelligent Parallelization of Java 8 Streams

Yiming Tang[1] Raffi Khatchadourian[1] Mehdi Bagherzadeh[2] Syed Ahmed[2]

[1]City University of New York (ponder@hunter.cuny.edu) [2]Oakland University

Introduction

The Java 8 Stream API sets forth a promising new programming model that incorporates functional-like, MapReduce-style features into a mainstream programming language.

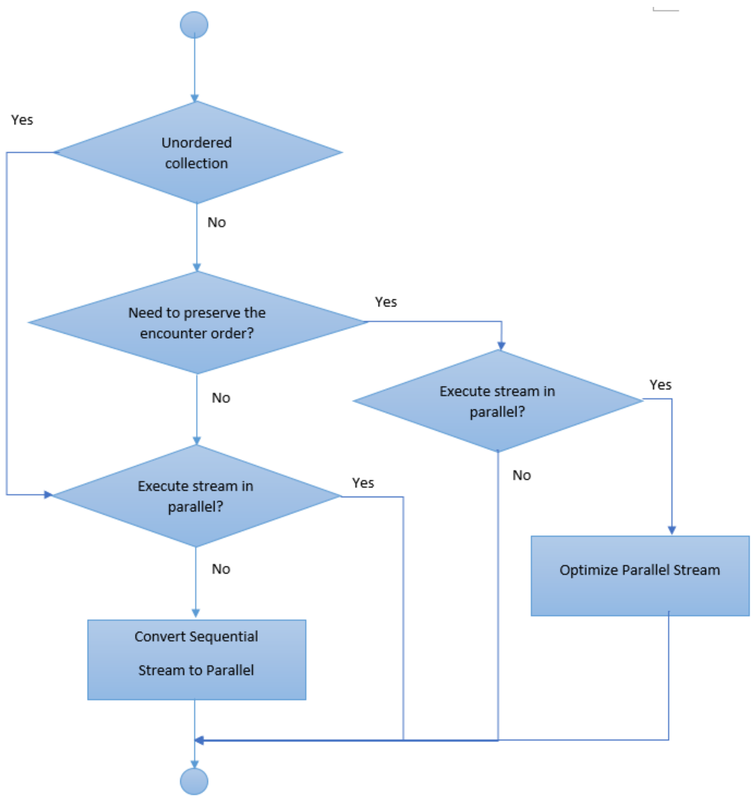
Problem

- Developers must *manually* determine whether running streams in parallel is efficient yet interference-free.
- Using streams correctly and efficiently requires many subtle considerations that may not be immediately evident.
- Manual analysis and refactoring can be error- and omission-prone.

Automated Tool

Our Eclipse Plug-in, based on ordering and typestate analysis, automatically identifies and executes refactoring opportunities where improvements can be made to Java 8 Stream code. The parallelization is "intelligent" as it carefully considers each context and may actually result in de-parallelization.

Flowchart



Contributions

We devise an automated refactoring approach that assists developers in writing optimal stream code. The approach determines when it is safe and advantageous to convert streams to parallel and optimize parallel streams. A case study is performed on the applicability of the approach.

Refactorings

1. Convert Sequential Stream to Parallel. Determines if it is advantageous and safe to convert a sequential stream to parallel.
2. Optimize Parallel Stream. Decides which transformations can improve the performance of a parallel stream, including unordering and converting to sequential.

Code Snippet of Widget Collection Processing Using the Java 8 Steam API

```
1  Collection<Widget> unorderedWidgets =
2    new HashSet<>();
3  List<Widget> sortedWidgets =
4    unorderedWidgets
5      .stream()
6      .sorted(Comparator.comparing(
7        Widget::getWeight))
8      .collect(Collectors.toList());
9  Collection<Widget> orderedWidgets =
10   new ArrayList<>();
11 Set<Double> distinctWeightSet =
12   orderedWidgets
13     .stream().parallel()
14     .map(Widget::getWeight).distinct()
15     .collect(Collectors.toCollection(
16       TreeSet::new));
```

(a) Stream code snippet prior to refactoring.

```
1  Collection<Widget> unorderedWidgets =
2    new HashSet<>();
3  List<Widget> sortedWidgets =
4    unorderedWidgets
5      .~~stream()~~parallelStream()
6      .sorted(Comparator.comparing(
7        Widget::getWeight))
8      .collect(Collectors.toList());
9  Collection<Widget> orderedWidgets =
10   new ArrayList<>();
11 Set<Double> distinctWeightSet =
12   orderedWidgets
13     .stream().~~parallel()~~
14     .map(Widget::getWeight).distinct()
15     .collect(Collectors.toCollection(
16       TreeSet::new));
```

(b) Improved stream client code via refactoring.

Typestate Analysis

Our in-progress approach uses typestate analysis to determine stream attributes when a terminal operation is issued. A typestate variant is being developed since operations like `sorted()` return (possibly) new streams derived from the receiver with their attributes altered. Labeled transition systems (LTSs) are used for execution mode and ordering.

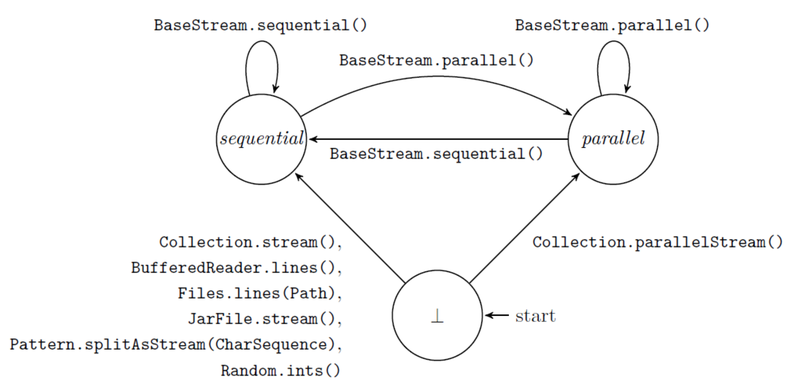


Figure: LTS for execution mode.

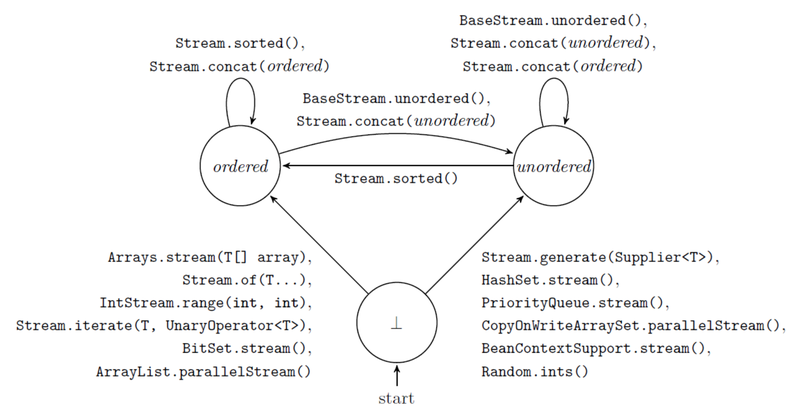


Figure: LTS for ordering.

Preliminary Experimental Results

projects	candidate streams	refactorable streams
experiments	1	0
threeten-extra	2	2
jOOQ	3	0
dari	4	0
JacpFX	4	3
bootique	5	0
jdk8-experiments	16	4
htm.java	21	7
jetty-project	22	7
streamql	22	2
java-design-patterns	28	17
Grand Total	128	42

Table: Preliminary results summary.

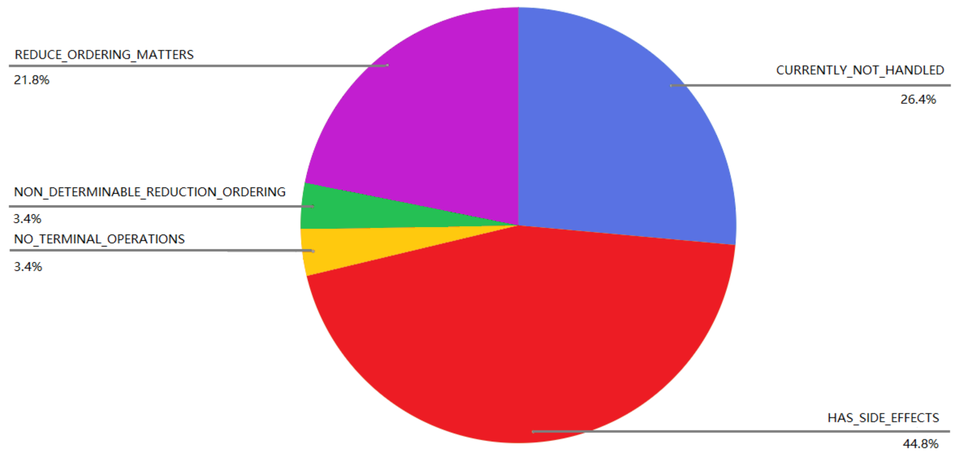


Figure: Refactoring precondition failures.

Conclusion

We have developed an automated refactoring approach that "intelligently" optimizes Java 8 stream code. Based on ordering and typestate analysis, it automatically deems when it is safe and advantageous to run stream code either sequentially or in parallel.

Future Work

- Expand our corpus.
- Handle several issues between Eclipse and WALA.
- Formulate a transformation algorithm.
- Incorporate additional reductions like those involving maps.

International Conference on Software Engineering, May 27–June 3, 2018, Gothenburg, Sweden